

Ergänzungen zu SWE2

Innere Klassen

Diverse neue Sprachfeatures bei Java 1.5

Generics



Inhalt

Innere Klassen

Diverse neue Sprachfeatures bei Java 1.5

Generics



Innere Klassen

- Innere Klassen werden innerhalb einer anderen Klasse definiert
 - lassen sich in der definierenden Klasse verbergen
 - haben Zugriff auf Daten und Methoden der definierenden Klasse
 - können „anonym“, d.h. ad hoc und ohne Klassennamen, definiert werden
- Es gibt vier Typen von inneren Klassen
 - Statische Member-Klassen
 - Member-Klassen
 - Lokale Klassen
 - Anonyme Klassen



Typen von inneren Klassen

- Statische Member-Klassen
 - werden `static` deklariert
 - sind wie Toplevel-Klassen
 - erreicht aber nur `static`-Members der definierenden Klasse
- Member-Klassen
 - werden als Member der Klasse definiert
 - Instanzen der Member-Klasse gehören immer zu einer Instanz der definierenden Klasse
 - Code der Member-Klasse hat Zugriff auf Code der umgebenden Klasse
- Lokale Klassen
 - werden innerhalb eines Blocks definiert
 - nur innerhalb des Blocks sichtbar
 - haben Zugriff auf den Code der definierenden Klasse
 - haben Zugriff auf alle innerhalb des Blocks bekannten final Variablen
- Anonyme Klassen
 - lokale Klasse ohne Namen



Statische Member-Klassen

- `static` definiert
 - analoge Eigenschaften wie andere `static`-Member einer Klasse
- selbstständige Klasse innerhalb des „Namensraums“ der definierenden Klasse
 - wird über definierende Klasse angesprochen:
Definiert als: `StatischeMemberKlasse`
- hat Zugriff auf alle `static`-Members der definierenden Klasse
- Zugriffsmodifikatoren bei statische Member-Klassen wichtig (`private`, `public`, ...)
- auch Definition eines Interfaces möglich

Anwendung:

- um eine Klasse innerhalb der definierenden Klasse zu verbergen (`private`)
- um eine Klasse innerhalb des Namensraums der definierenden Klasse zu definieren (`public`)



Beispiel statische Member-Klasse: `Linkable` in `LinkedList`

- Interface `Linkable` ist eine statische Member-Klasse, die zu `LinkedList` gehört

```
public class LinkedList {  
  
    public static interface Linkable {  
        public Linkable getNext();  
        public void setNext(Linkable node);  
        public Object getValue();  
    }  
  
    public void push(Linkable node) {...}  
    public Object pop() {...}  
}  
  
class LinkableInteger implements Linkable {  
    public Linkable getNext() {...}  
    public void setNext(Linkable node) {...}  
    public Object getValue() {...}  
    ...  
}
```



Member-Klasse

- als nicht-statische Member der Klasse definiert
- Objekte gehören immer zu Objekt der definierenden Klasse
= **umgebendes Objekt**
 - Instanzen der Member-Klasse müssen immer mit umgebenden Objekt erzeugt werden
 - Code der Member-Klasse hat Zugriff auf Instanzfelder und -methoden des umgebenden Objekts

Anwendung:

- wenn ein Objekt eng mit dem umgebenden Objekt verbunden werden soll



Beispiel Member-Klasse: Linkable in LinkedStack

- Enumerator als Member-Klasse von LinkedStack arbeitet direkt mit einem LinkedStack-Objekt

```
public class LinkedStack {  
  
    private Linkable head;  
    ...  
  
    public java.util.Enumeration enumerate() {  
        return this.new Enumerator();  
    }  
  
    protected class Enumerator implements java.util.Enumeration {  
        Linkable current;  
        public Enumerator() { current = head; }  
        public boolean hasMoreElements() { return (current != null); }  
        public Object nextElement() {  
            if (current == null) { throw new java.util.NoSuchElementException(); }  
            Object value = current.getValue();  
            current = current.getNext();  
            return value;  
        }  
    }  
}
```



Objekterzeugung bei Member-Klassen

- Bei Objekterzeugung von Member-Klassen mit `new`, muss umgebendes Objekt angegeben werden
- Erfolgt durch Angabe des Objektzeigers auf das umgebende Objekt beim `new`-Operator

```
LinkedList stack = new LinkedList();  
Enumeration e = stack.new Enumerator();
```

```
...  
public java.util.Enumeration enumerate() {  
    return this.new Enumerator();  
}
```

Anmerkung:

`this` kann weggelassen werden (wie im Beispiel auf der letzten Folie)



Lokale Klassen

- lokal in einem Block definiert
- auf den definierenden Block beschränkt
- sonst ähnlich Member-Klasse, d.h. gehören zu einem umgebenden Objekt und haben gleiche Zugriffsmöglichkeiten
- kann auf lokale Variablen, die als `final` deklariert wurden, zugreifen

Anwendung:

- wie Member-Klassen, aber die Gültigkeit der Klasse soll auf Block eingeschränkt bleiben



Beispiel lokale Klasse: Enumerator lokal in enumerate

- Enumerator wird lokal im Block der Methode enumerate definiert

```
public class LinkedList {  
  
    private Linkable head;  
    ...  
  
    public java.util.Enumeration enumerate() {  
  
        protected class Enumerator implements java.util.Enumeration {  
            Linkable current  
            public Enumerator() { current = head; }  
            public boolean hasMoreElements() { return (current != null); }  
            public Object nextElement() {  
                if (current == null)  
                    throw new java.util.NoSuchElementException();  
                Object value = current.getValue();  
                current = current.getNext();  
                return value;  
            }  
        }  
  
        return new Enumerator();  
    }  
}
```



Anonyme Klassen

- Lokale Klasse ohne Namen
- werden mit einem new-Operator definiert, der das Objekt anlegt
- Klasse für ein einziges Objekt

Anwendung:

- für nur ein Objekt benötigt
- für kleinen Adapter-Code
- insbesondere für die Realisierung von Listener beim JavaBeans-Ereigniskonzept (siehe JavaBeans)



Beispiel anonyme Klassen: MouseAdapter innerhalb MyFrame

- anonyme Spezialisierung von MouseAdapter
- wird als MouseListener der Komponente table angefügt
- überschreibt Methode mouseClicked

```
public class MyFrame extends JFrame {
    ...

    public MyFrame() {
        table = new.JTable();

        table.addMouseListener(
            new MouseAdapter() {
                public void mouseClicked(MouseEvent e) {
                    if (e.getClickCount() >= 2) {
                        editAppointment();
                    }
                }
            });
    }
}
```



Beispiel anonyme Klassen: MouseAdapter innerhalb MyFrame

- anonyme Spezialisierung von MouseAdapter
- wird als MouseListener der Komponente table angefügt
- überschreibt Methode mouseClicked

```
public class MyFrame extends JFrame {
    ...

    public MyFrame() {
        table = new.JTable();

        class MyMouseAdapter extends MouseAdapter {
            public void mouseClicked(MouseEvent e) {
                if (e.getClickCount() >= 2) {
                    editAppointment();
                }
            }
        }
        table.addMouseListener(new MyMouseAdapter());
    }
}
```



Beispiel anonyme Klassen: MouseAdapter innerhalb MyFrame

- anonyme Spezialisierung von MouseAdapter
- wird als MouseListener der Komponente table angefügt
- überschreibt Methode mouseClicked

```
public class MyFrame extends JFrame {
    ...

    public MyFrame() {
        class MyMouseListener extends MouseAdapter {
            public void mouseClicked(MouseEvent e) {
                if (e.getClickCount() >= 2) {
                    editAppointment();
                }
            }
        }

        table = new JTable();

        table.addMouseListener(new MyMouseListener());
    }
}
```



Inhalt

Innere Klassen

Diverse neue Sprachfeatures bei Java 1.5

Generics



Boxing und Unboxing

Nur Referenztypen sind mit *Object* kompatibel

Referenztypen: Klassen, Arrays, Interfaces

Werttypen: int, double, char, ...

Java 1.4: Werttypen müssen über Wrapper kompatibel zu *Object* gemacht werden

```
Object obj = new Integer(3);
Integer i = (Integer) obj;
int val = i.intValue();
```



```
class Integer {
    int val;
    ...
}
```

Java 1.5: Wrapper werden automatisch erzeugt

```
Object obj = 3; // (auto) boxing
int x = (Integer) obj; // (auto) unboxing
```

```
Object obj = 3.14; // (auto) boxing
double x = (Double) obj; // (auto) unboxing
```

```
Object obj = 'x'; // (auto) boxing
char x = (Character) obj; // (auto) unboxing
```



Anwendung in Collection-Klassen

```
ArrayList list = new ArrayList();
list.add(10);
list.add(20);
int x = (Integer) list.get(0);
```

formaler Parameter ist vom Typ *Object*

liefert Wert vom Typ *Object*

Cast kann mittels Generics vermieden werden (siehe später)

```
Hashtable phone = new Hashtable();
phone.put("Meier", 12345);
phone.put("Mueller", 34567);
int number = (Integer) phone.get("Meier");
```

beide formalen Parameter sind vom Typ *Object*

liefert Wert vom Typ *Object*



Enumerationstypen

Java 1.4: Menge benannter Konstantenwerte

```
// colors
static final int RED = 0;
static final int BLUE = 1;
static final int GREEN = 2;
```

```
int color = BLUE;
```

```
// priorities
static final int LOW = 0;
static final int NORMAL = 1;
static final int HIGH = 2;
```

```
int priority = HIGH;
```

Problem: keine Typprüfung zwischen Konstantenmengen

```
int color = HIGH; ← Compiler meldet keinen Fehler
```

Java 1.5: Enumerationstypen

```
enum Color {
    RED, BLUE, GREEN
}
```

```
Color color = Color.BLUE;
```

Compiler prüft, daß nur
Color-Werte zugewiesen
werden

```
enum Priority {
    LOW(1), NORMAL(2), HIGH(4);
    private int val;
    private Priority(int val) { this.val = val; }
    public int value() { return val; }
}
```

```
Priority prio = Priority.HIGH;
System.out.println(prio.value()); // 4
```



Neue for-Schleife

Java 1.4: Iterieren über Arrays und Collections

```
int[] primes = {2, 3, 5, 7, 11};
for (int i = 0; i < primes.length; i++) {
    System.out.println(primes[i]);
}
```

```
ArrayList names = new ArrayList();
names.add("Alice");
names.add("Bob");

Iterator iter = names.iterator();
while (iter.hasNext()) {
    System.out.println((String)iter.next());
}
```

Java 1.5: Neue for-Schleife

```
int[] primes = {2, 3, 5, 7, 11};
for (int x: primes) {
    System.out.println(x);
}
```

spricht:
for each int x in primes

```
ArrayList names = new ArrayList();
names.add("Alice");
names.add("Bob");

for (Object s: names) {
    System.out.println((String)s);
}
```



Variable Anzahl von Parametern

Java 1.4: Methoden mit fixer Parameteranzahl

```
static int sum(int x, int y) { return x + y; }
```

```
int x = sum(17, 4);
```

← kann nur die Summe von 2 Zahlen bilden

Java 1.4: Methoden mit variabel vielen Parametern vom gleichen Typ

```
static int sum(int[] val) {  
    int res = 0;  
    for (int i = 0; i < val.length; i++) res += val[i];  
    return res;  
}
```

```
int x = sum(new Integer[] {1, 2, 3, 4, 5});
```

Java 1.5: "vararg-Parameter"

```
static int sum(int... val) {  
    int res = 0;  
    for (int i = 0; i < val.length; i++) res += val[i];  
    return res;  
}
```

← vararg-Parameter
muß der letzte formale Parameter sein

```
int x = sum(1, 2, 3, 4, 5);
```



Inhalt

Innere Klassen

Diverse neue Sprachfeatures bei Java 1.5

Generics



- Siehe Skriptum SWE 2

